Complex Adaptive Systems Modeling
a SpringerOpen Journal

RESEARCH Open Access

Complex adaptive systems modeling with Repast Simphony

Michael J North[1,2*], Nicholson T Collier[1], Jonathan Ozik[1,2], Eric R Tatara[1], Charles M Macal[1,2], Mark Bragen[1] and Pam Sydelko[1]

* Correspondence: north@anl.gov
[1]Decision and Information Sciences Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439-4867, USA
[2]University of Chicago, Chicago, IL, USA

Abstract

Purpose: This paper is to describe development of the features and functions of Repast Simphony, the widely used, free, and open source agent-based modeling environment that builds on the Repast 3 library. Repast Simphony was designed from the ground up with a focus on well-factored abstractions. The resulting code has a modular architecture that allows individual components such as networks, logging, and time scheduling to be replaced as needed. The Repast family of agent-based modeling software has collectively been under continuous development for more than 10 years.

Method: Includes reviewing other free and open-source modeling libraries and environments as well as describing the architecture of Repast Simphony. The architectural description includes a discussion of the Simphony application framework, the core module, ReLogo, data collection, the geographical information system, visualization, freeze drying, and third party application integration.

Results: Include a review of several Repast Simphony applications and brief tutorial on how to use Repast Simphony to model a simple complex adaptive system.

Conclusions: We discuss opportunities for future work, including plans to provide support for increasingly large-scale modeling efforts.

Keywords: Complex adaptive systems modeling, Agent-based modeling, Java, Groovy

Background

Introduction

Complex adaptive systems (CASs) are composed of interacting, autonomous agents (Holland 2006). CAS agents have properties and behaviors. They interact with and influence each other, learn from their experiences, and adapt their behaviors so they are better suited to their environment(s). By modeling these agents individually, the full effects of the diversity that exists among agents with respect to their attributes and behaviors can be observed as they give rise to the dynamic behavior of the system as a whole.

Agent-based modeling provides a mechanism for modeling CASs (Bonabeau 2002; Macal and North 2010). Agent-based modeling has been used successfully to model complex adaptive systems in many disciplines, including archaeology, biology, ecology, supply chains, consumer market analysis, military planning, and economics (North and Macal 2007; North and Macal 2009). Some examples are reviewed in section three.

Springer

© 2013 North et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Repast Simphony is a widely used, free, and open source environment for agent-based modeling of CAS. The most recent version, 2.0, was released on March 5, 2012. Repast Simphony is a second-generation environment that builds upon the previous Repast 3 library described in North et al. (2006).

This paper describes Repast Simphony and shows how to apply it to modeling CASs. The next section discusses related work. Section three details the architecture of Repast Simphony. Section four uses a simple demonstration example to show how to model CAS with Repast Simphony. Conclusions are then presented.

Related work

This section reviews other free and open source modeling libraries and environments that are documented with published papers. The review includes NetLogo, StarLogo, Swarm, Repast 3, MASON, Ascape, and EcoLab.

StarLogo

StarLogo (StarLogo 2011) is a library and environment that uses a Java interpreter and interface. StarLogo is an educational system that leverages the Logo language (Feurzeig et al. 1970, Harvey 1997) to make it easier to learn the library and to use it to develop agent models. StarLogo is free and open source for non-commercial use. StarLogo substantially extended Logo by increasing the allowed number of agents (i.e., turtles) from tens to thousands by giving turtles much more interactive behaviors, and by transforming the turtle's environment from a rectangular array of pixels into a grid of first-class agents (i.e., patches). Resnick (1996) notes that StarLogo is designed around a single organizing principle. For StarLogo, that principle is modeling decentralized complex systems. There is no requirement to hide the complexity of foundational features, such as parallelism, if they embody the organizing principle.

NetLogo

NetLogo (Wilensky 2012) is a free and open source agent-based simulation environment that uses a modified version of the Logo programming language. NetLogo is available under the General Public License (GPL) (NetLogo User Manual – FAQ (Frequently Asked Questions)). NetLogo's builds on the StarLogo effort (Tisue and Wilensky 2004). NetLogo was designed to provide a basic laboratory for teaching complexity concepts. It can also be used to develop more sophisticated applications. NetLogo provides a graphical environment to create programs that control graphic turtles that reside in a world of patches, which are monitored by an observer. Links are also available to connect turtles to form networks. Wilensky (2012) and Tisue and Wilensky (2004) offer several lessons from the development of NetLogo. Agent-based simulation environments and languages should be simple enough to allow beginners to start quickly. Agent modeling libraries should, in principle, place no limits on what experienced users can do. Agent modeling libraries should also include a large number of example simulations to help beginning and experienced users alike.

For Wilensky (2012) and Tisue and Wilensky (2004), programming is not something to be avoided. The challenge is not to develop a programming-free way to specify models. Most people can learn to program. The challenge is to simplify programming syntax and semantics enough to make programming accessible. Visual programming is not

necessarily the easiest way to specify models. They state that working with a carefully designed text-based language can sometimes be easier than a visual one.

Swarm

Swarm is a free and open source toolkit with both Objective-C and Java bindings (Minar et al. 1996). Swarm also uses a GPL license. Swarm was originally developed as a software toolkit for the creation of simulation models in the field of Artificial Life (A-Life) (Macal 2009). Swarm provides modelers with a flexible, nested approach to modeling interactions. Swarm was highly influential in the development of libraries that would arrive on the scene later, such as Repast 3 (North et al. 2006) and Repast Simphony. While discussing their design, (Minar et al. 1996) states that agent-based modeling toolkits or libraries are important in that they save scientists from wasting time on repetitive and error-prone programming. Discrete time scheduling is preferred over floating point scheduling, because discrete calculations do not suffer from rounding errors. Object orientation is also useful because of its close association with agents.

Minar et al. (1996) suggests that agents should be organized into containers with schedules of activities. These collections are called "swarms." Swarms should be interoperable to allow mixing and nesting of related models.

Repast 3

Repast 3 is a family of three free and open source agent-based modeling libraries (North et al. 2006). The three libraries are Java-based Repast J, C#-based Repast .NET, and NQP (Not Quite Python)-based Repast Py. Repast 3 uses a "new BSD" (Berkeley Software Distribution)-style license and includes third-party libraries with compatible licenses. Several conclusions were reached during the implementation of the libraries. Object-oriented design is essential to allow both the library and user models to be flexible. The use of design patterns is helpful for improving implementation quality. Environments should allow seamless programming in different languages to help simultaneously lower barriers to entry for new users and increase flexibility for experts. Choosing environments that are cross-library makes systems usable for a wider audience compared to single library approaches. Modular construction is essential. Library developers should be keenly aware of differences in the fundamental details of the candidate implementation environments, such as differences in mathematics libraries. Library developers should also consider the availability and capability of the ecosystem surrounding an implementation environment when making their selections.

MASON

MASON is a free and open source, agent-based modeling library that provides core services that can be mixed easily with other libraries (Luke et al. 2005). Luke et al. (2005) recommends that libraries should be developed from first principles derived from specific domains. Another key attribute is that libraries should be minimal and mix well with third-party libraries. Only essential features should be provided to increase interoperability and simplify learning. There should be complete separation between models, user interfaces, and offline storage. A large number of agents (e.g., millions) should be supported. All model runs should be reproducible. Furthermore, checkpointing (i.e., saving and restoring model runs midstream) should be supported.

Ascape

Ascape (Parker 2001) is a free and open source, agent-based modeling library that represents models as a complex series of nested "scapes" populated by agents whose behaviors are implemented with abstracted rules. Parker (2001) states that the abstraction of complexity is a powerful way to focus design and development efforts. Selective abstraction should be used to provide structure for users and reduce the complexity of user code. The abstractions need to be very carefully considered to avoid limiting flexibility.

For Parker (2001), space should be represented with nested containers (i.e., scapes). Scapes dynamically represent space, identity type, and group membership. Scapes should be allowed to act as agents. This treatment allows complete nesting of models. Agents, however, should not have to explicitly address the nature of scapes, which greatly simplifies agent specifications and reduces the chances for errors. Agent behavior specification and scheduling should occur only at the scape-level, never at the individual level. This treatment allows behaviors to be dynamically changed and permits sophisticated, behavior-based designs of experiments.

EcoLab

EcoLab (Standish 2008) is a general purpose C++ and TCL/Tk agent-based modeling library. Users write models in C++ and then invoke the models using TCL. Tk can be used as needed to develop user interfaces. Standish (2008) suggests that it is helpful to have standardized models with which to compare agent modeling libraries. Standish states that reflection is essential for agent models. Reflection is the ability to examine the type information of objects dynamically, that is, while a program is running. Standish also states that dynamic pointers that automatically track objects such as agents are important. These pointers should automatically note the type of object being tracked and should indicate whether an object reference has been deallocated from memory.

Methods

Repast Simphony

The free and open source REcursive Porous Agent Simulation Toolkit (Repast) was originally developed by Sallach, Collier, and others (Collier et al. 2003) at the University of Chicago in 2000, and was subsequently expanded by Argonne National Laboratory as a reusable software infrastructure that could support "rapid social science discovery" based on extensive computational experimentation (Sallach and Macal 2001). Successive releases of Repast have extended the system to handle large-scale agent simulation application development. Repast is managed by the not-for-profit volunteer Repast Organization for Architecture and Design (ROAD). ROAD is led by a board of directors that includes members from a range of government, academic, and industrial organizations. Argonne National Laboratory is especially active in maintaining and developing Repast. Many other groups contribute to Repast's development. For example, the University of Michigan recently teamed with Argonne to add more than 20 demonstration models to Repast Simphony with support from the 2011 Google Summer of Code. The Repast system, including the source code, is available directly from the Web at http://repast.sourceforge.net/ (Repast – The Repast Suite). All versions of Repast use a "new BSD"-style license and include third-party libraries with compatible licenses.

Repast Simphony builds on the lessons of Repast 3 but uses a new code base. Repast Simphony was designed from the ground up with a strong focus on well-factored abstractions. The resulting code has a highly modular plug-in architecture that allows individual components such as networks, logging, and time scheduling to be replaced as needed. A plug-in is a software module that extends the features of a host program using a well-defined programming interface. The Repast Simphony code also has multiple layers within each plug-in module to allow specific implementations at lower levels to be replaced by new implementations when required. As discussed later, this layering allowed the authors to replace Repast Simphony's logging and visualization implementations while maintaining backward compatibility. Repast Simphony also includes many advanced features, such as watchers, that were not possible in Repast 3. These features are discussed later.

The Repast community is large and growing. These users have applied Repast to a wide variety of applications that range from social systems, to evolutionary systems, market modeling, and industrial analysis. A sampling of Repast applications will be presented next.

Example Repast Simphony applications

This section briefly discusses several example Repast Simphony applications that illustrate the versatility of the system. The examples include applications that study phenomena related to consumer products, systems engineering, possible future hydrogen infrastructures, social science, and ancient pedestrian traffic, to name a few. Additional examples can be found in North and Macal (2009).

The Virtual Market Learning Lab (North et al. 2010) is a large-scale, Repast Simphony model of consumer markets co-developed by Argonne National Laboratory and Procter & Gamble (P&G). It represents the shopping behavior of consumer households and the business behavior of retailers and manufacturers in a simulated national consumer market. Argonne and P&G successfully calibrated, verified, and validated the resulting agent-based model using several independent, real-world data sets for multiple consumer product categories with more than 60 comparison tests per data set. First Repast and then later Repast Simphony were used to implement the model. P&G has successfully applied the model to several challenging business problems where it has directly influenced managerial decision-making and has produced substantial cost savings.

The Monitoring, Analysis, and Diagnosis of Distributed Processes with Agent-Based Systems (MADCABS) model is a Repast Simphony simulation framework for investigating the operation and control of complex networks (Artel et al. 2011; Tatara et al. 2006). MADCABS combines agent models with traditional differential equations models. MADCABS has been used for a variety of research pursuits, including to study the effectiveness of perceptron-based learning for the robust operation of distributed chemical reactor networks.

The Hydrogen Economy Model (Mahalik et al. 2007) is a Repast Simphony representation of the Los Angeles, California, metropolitan area, with 5,000 square miles of detailed and geographic information system (GIS)-sourced interstate highways and omnipresent local roads. The model has driver and investor agents. Driver agents use their cars to move between their demographically assigned home neighborhoods and their jobs. Drivers have a variety of characteristics including income levels, environmental concerns, risk

aversion, and car type preferences (e.g., wanting conventional fuel cars versus hydrogen cars). Investor agents build, own, and operate hydrogen fuel stations based on the investor's estimates of the potential for profits at each available geolocated site.

The Endogenous Emergence of Coordination (EndEC) model (Ozik and North 2010) is a Repast Simphony simulation of the endogenous emergence of coordination within a group of social agents. The model demonstrates the use of a dynamic language, specifically Groovy, to represent the socially coordinated evolution of behavior. The EndEC sample model is inspired in part by Holland's work on the emergence of language (Ke and Holland 2006; Tao et al. 2005). The model consists of a set of agents each with an individual list of movement capabilities and utterances. The agent's movement capabilities and utterances are drawn from separate supersets that are available to the population as a whole. The agents each maintain their own individual associations of movements with utterances. As the simulation proceeds, agents make utterances to announce their movements and, over time, adjust their associations to match the observed behavior of their neighbors. These simple behavior rules result in the dynamic emergence of complex coordination between groups of agents.

SHULGI is a Repast Simphony model of ancient pedestrian traffic in urban environments (Branting et al. 2007). The model represents a wide range of issues that affect pedestrian route choices, including source and destination locations; roadway availability and quality; and the metabolic energy costs of candidate routes. The factors can be flexibly weighted as needed to address different questions of interest. The model has been successfully validated by using archaeological data from the ancient Turkish city of Kerkenes Dag.

Repast Simphony design goals

The design goals of Repast Simphony are as follows (North et al. 2005):

- All of the core features and capabilities of the previous versions of Repast should be available. These features are detailed in North et al. (2006). The lessons learned from developing the original Repast series of libraries are summarized above in the related work section.
- There should be a strict separation between model specification, model execution, model visualization, and data storage.
- All user model components should be plain, unadorned Java objects that are accessible to and replaceable with external software (e.g., legacy models and enterprise information systems).
- Tasks that are commonly performed by model developers should be automated when possible.
- Imperative 'boilerplate' code should be eliminated or replaced with declarative runtime configuration settings when possible. Much like boilerplate text in standard English, boilerplate code is moderate- to large-sized pieces of code that are used again and again in model after model. An example in Repast 3.1 is code to setup a master time schedule or configure a network display.
- Idiomatic code expressions should be simple and direct. Idiomatic code expressions are short pieces of code that act as slang terms or shorthand. An example is code to update the weight on a network edge between agents.

These design goals have been met by the Repast Simphony architecture.

Eclipse

Eclipse (Eclipse Home Page) is a widely used, free, and open source development environment that is multilingual and integrated. Eclipse can be used to develop code in many languages, including Java and C++. Java support is particularly strong.

The Repast Simphony system uses Eclipse as its primary development environment. Repast Simphony leverages Eclipse's plug-in architecture to provide a set of development options. It is important to note that while Eclipse's plug-in architecture is conceptually similar to that of Repast, Eclipse uses a different implementation.

The Repast Simphony Eclipse plug-ins provide tools, views (i.e., windows), and perspectives (i.e., selected sets of windows) for creating a range of Repast-specific model components, including general Repast projects, ReLogo projects, flowcharts, and ReLogo agents. The plug-ins also allow Repast models to be executed, debugged, and packaged into self-contained installers for deployment. An example flowchart that can be used to encode agent attributes and behaviors is shown in Figure 1.

Repast Simphony architecture

The design goal of maintaining a strict separation between model specification, model execution, data storage, and visualization has led to creation of the multilingual Repast Simphony model development environment in Eclipse, a runtime engine for model execution, a runtime interface for model visualization, and freeze dryers for data storage. Freeze dryers are an automated way to send and retrieve the state of a simulation to and from secondary storage, such as an XML (extensible markup language) file. To implement this function, Repast Simphony uses a highly modular architecture. Each module is an independent 'plug-in' that can be connected or disconnected with a few

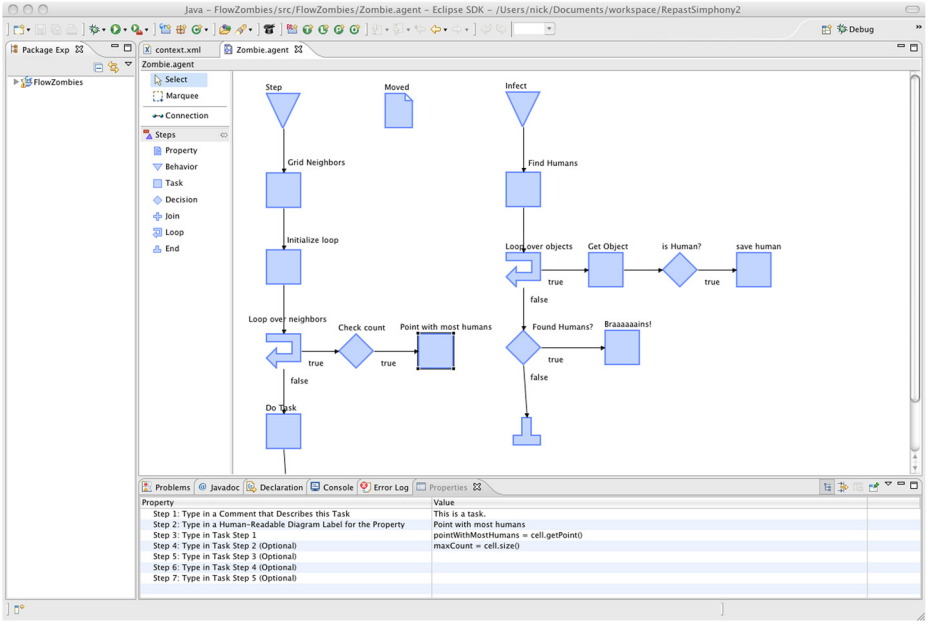

Figure 1 An example Repast Simphony flowchart showing the behavior for a Zombie agent.

lines of XML. Thanks to this architecture, Repast Simphony exists, in essence, as a user-selectable federation of plug-in components. Selected plug-ins and related sets of plug-ins are summarized in Table 1.

It should be noted that that Repast plug-ins use the Java Plug-in Framework (JPF) (Java Plug-in Framework (JPF) Project), which is separate from the Eclipse plug-in architecture. JPF is used at runtime by Repast models. Eclipse is used at development time by the Repast design tools.

The next section will discuss the plug-in system. The following sections will consider central plug-ins of particular interest.

Simphony application framework

Repast Simphony's modular architecture is built on top of the free and open source Simphony Application Framework (SAF). As an application framework, SAF is intended to simplify common application-building tasks, such as creating and configuring menus, toolbars, application window layout, and so on. SAF-based applications typically follow the model-view-controller design pattern with the addition of a class that mediates between various parts of the application. This mediator provides a link between the user interface and the application functionality triggered by the user. The Repast Simphony runtime interface implementation consists of three components: the user interface proper, the simulation engine infrastructure, and the application class that mediates the interaction between the two. The user interface is composed of other smaller model-view

Table 1 Selected repast simphony plug-ins and plug-in sets

Plugin or plugin sets	Function
Simphony Application Framework	Provides the basic runtime interface plug-in system and user interface tools
Eclipse Set	Provides model specification and programming tools
Core Set	Provides central simulation functions, such as scheduling
ReLogo Set	Provides a simple modeling language and structure
GIS	Provides GIS modeling and visualization
Freeze Dryer Set	Provides persistence in XML and text formats
Batch Run Set	Provides parameter sweeps and stochastic iteration
Deployment	Packages user models for self-contained release
Charts	Provides interactive model graphing in the runtime interface
Model Integration	Provides tools for embedding legacy models
2D Visualization Set	Provides tools for interactive two-dimensional (2D) model viewing
3D Visualization Set	Provides tools for interactive three-dimensional (3D) model viewing
Terracotta	Provides distributed model execution
System Dynamics	Provides tools for ordinary differential and difference equations
Third-Party Application Set	Supports a set of independent plug-ins for: • Geographic Resources Analysis Support System (GRASS), • Java Universal Network/Graph Framework (JUNG) network analysis, • *ORA network analysis, • Pajek network visualization, • R statistics, • iReport and Jasper Reports enterprise reporting, • Spreadsheets, • Structured Query Language (SQL) analysis within running simulations, • VisAD scientific visualization, and • Weka data mining.

-controller types of components that may or may not delegate some functionality to the application class.

SAF uses JPF to provide much of this functionality. JPF provides a runtime engine that dynamically discovers and loads plug-ins. A plug-in within the SAF context is a structured component that describes itself to JPF using a manifest, which is an XML document with a specific schema. This file contains information identifying the plug-in; what other plug-ins this plug-in depends on, if any; where its code is located; and where any third-party code that it depends on is located. JPF does not provide the plug-in content, but rather a plug-in discovery mechanism, a plug-in format, and an application programming interface (API) for working with such. SAF then uses these pieces to create an application framework.

This plug-in approach helps to make SAF applications modular. Specific bits of application functionality can be defined in separate plug-ins and loaded at runtime. Repast Simphony as a whole is simply a collection of plug-ins, allowing bits and pieces of Repast to be used as required. The Repast Simphony runtime interface also leverages this modularity in a variety of ways. For example, the integration with third-party applications, such as the R statistical package, that appear in the toolbar is accomplished by using the plug-in mechanism. Third-party application integration is discussed later.

In addition to promoting modularity, JPF also makes applications extensible by using plug-in extension points and extensions. An extension point functions as a generic interface that defines some specific functionality and how to invoke that functionality in an abstract way. An extension is an implementation of an extension point. SAF defines extension points appropriate for building applications, and the Repast Simphony runtime interface implements these as extensions. Both the Repast runtime interface and Repast Simphony define Repast-specific extension points and extension implementations. For example, the Repast Simphony runtime interface scenario tree that appears on the lower left side of Figure 2 contains various metadata (e.g., context loading, display definition, etc.) about the model and is itself a collection of extensions. In addition, the mechanisms by which these are manipulated (e.g., display wizards) are also extensions implementing extension points defined by the Repast Simphony runtime interface. In this way, Repast Simphony and the Repast runtime interface are modular and easily extensible. The value of Repast Simphony's modularity was demonstrated when the initial library used by SAF to provide docking windows needed to be replaced to meet increasing user requirements. The modular structure allowed the docking windows library to be swapped out while maintaining backward compatibility with existing user models.

Core

The Repast Simphony core plug-ins provide major simulation functions, including time scheduling, space management, behavior activation, and random number generation (North et al. 2007). Repast Simphony space management uses contexts and projections. Repast Simphony provides a discrete event time scheduler that uses double precision real numbers for event times. The scheduler also allows events with otherwise identical times to be differentiated with an optional priority ranking. The Repast scheduler offers imperative scheduling and declarative scheduling for both sequential and concurrent activities, as discussed later in this section.

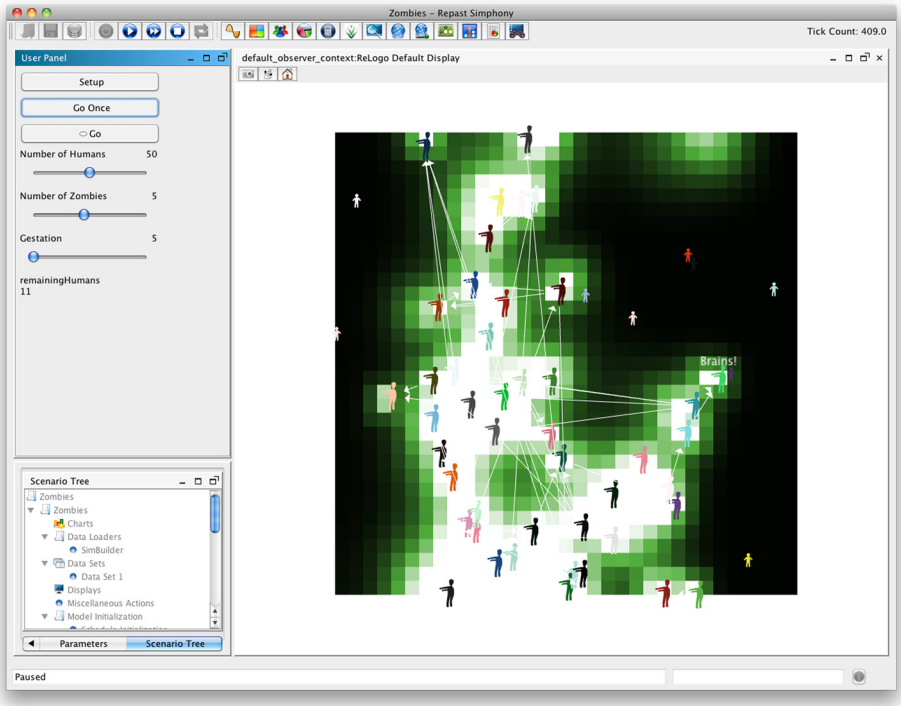

Figure 2 The Repast Simphony zombie demonstration model.

Repast Simphony contexts are hierarchically nested and named containers that hold model components. The model components can be any type of Java object, including other contexts, but are typically agent objects. It should be noted that contexts can be agents as well. Each model component can be present in as many contexts as the modeler desires. The hierarchical nesting means that a model component that is present in a context is also present in all of that context's parent contexts. Of course, the converse is not true in the general case. The hierarchical nesting structure can be declaratively or imperatively specified by the modeler. In addition, the contents of components within contexts (e.g., agent properties) can be declaratively logged at runtime.

In addition to supporting hierarchical nesting, contexts also support projections. Repast Simphony projections are named sets of relationships defined over the constituents of a context. For example, a Repast Simphony network projection stores a network or graph relationship between the members of its context. The members of this context can then ask to whom they are linked and who is linked to them. Similarly, the Repast Simphony grid projection stores a set of Cartesian coordinates for each member of the context. The members of this context can ask where they are located. Each context can support any mixture of projections. Also, projections can be declaratively visualized at runtime.

Contexts and projections can use watchers to activate behaviors. The following example shows an agent behavior that is activated at certain times when an appropriate social network neighbor changes its "contentment" attribute:

```
// This is an example watcher activation method.
@Watch(watcheeClassName =
```

```
    "repast.user.models.ExampleSocialAgent",
    watcheeFieldName = "contentment",
    query = "linked_to 'neighbors' and not linked_from
      'work' and colocated",
    triggerCondition = "$watchee.getContentment() !=
     $watcher.contentment",
    whenToTrigger = WatcherTriggerSchedule.LATER,
    scheduleTriggerDelta = 5,
    scheduleTriggerPriority = 15)
  public void neighborMoodUpdated(ExampleSocialAgent
    neighbor) {
    // Check the contentment level.
    if (neighbor.getContentment() > 0.5) {
      // Copy a neighbor.
      this.setContentment (neighbor.getContentment());
    } else {
     // Decide for ourselves.
     this.setContentment(RandomHelper.nextDouble());
    }
  }
```

In this example, the "neighborMoodUpdated" method will be called after another ExampleSocialAgent's contentment changes if the current agent is linked to the watched agent in the neighbor network, is not linked from the watched agent in the work network, is colocated with the watched agent in a context, and has a contentment setting different from that of the watched agent's contentment. The "neighborMoodUpdated" method will be called five ticks after the change and the event will be given a scheduler priority of 15, which is substantially higher than the default value of zero. As shown in the example, contexts work directly with watchers by allowing watcher queries to use context names and properties. Similarly, projections work directly with watchers, allowing watcher queries to use projection names (e.g., "neighbors" and "work" in Figure 3), properties, and relationships (e.g., "linked_to," "linked_from," and "colocated" in Figure 3).

The watcher mechanism provides a listener-like design pattern for reactive agent behavior, which is computationally more efficient than polling a collection of agents at a fixed interval. Runtime agent class instrumentation is performed with the Javassist library (Chiba and Nishizawa 2003) when the agent classes are loaded by the Repast runtime, which occurs automatically for classes with @Watch annotations. Bozada et al. (2006) presented one of the first implementations the Repast watcher pattern in a large-scale, agent-based model of military deployment logistics. The execution speed of the logistics model implementation in Repast was more than an order of magnitude faster than the previous Java implementation with polling-based agent behaviors.

Annotations can also be used for tasks such as unconditional scheduling. In this usage, individual methods can be called at specified times without the need for explicit Java method calls to the Repast scheduling system. This approach has the potential to be both

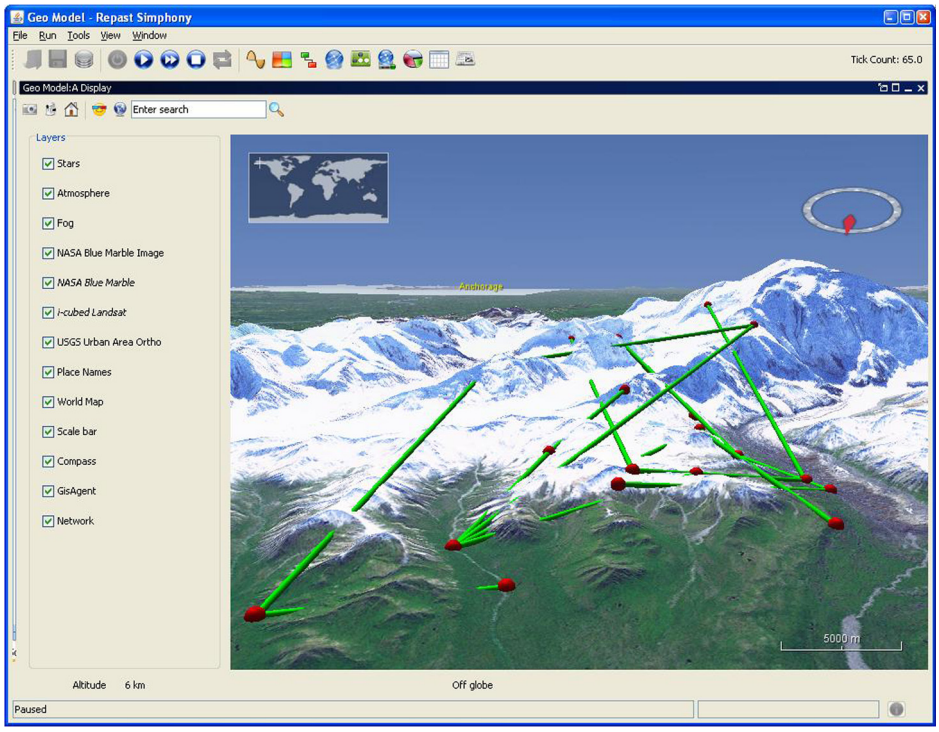

Figure 3 A Repast Simphony demonstration model that combines GIS with a network.

more convenient than the conventional Java method of call scheduling because it requires only the setting of a few annotation parameters rather than the making of a set of method calls. It is also better documented because it directly connects the scheduling information to the scheduled method itself rather than dividing this task among multiple locations in the model.

Both watch-based scheduling and unconditional scheduling allow concurrent or parallel code execution. This optional feature is activated by setting the duration of the activity being scheduled to a non-zero value. The duration is the length of time the activity is considered to take. Any activities that occur between the start of the activity with non-zero duration and the end of the duration are candidates to be concurrently executed with the activity having non-zero duration. The default duration is zero, indicating sequential execution. Whether the execution is simply concurrent or actually parallel is determined by the underlying hardware, the Java virtual machine in use, and the interrelated details of the activities being scheduled. This approach allows modelers to begin with simpler sequential execution and then scale up to concurrent or parallel execution as they continue to develop their models.

Many agent behaviors use random number generation for option selection. The Repast Simphony core module includes a full suite of random number generation tools. Repast uses the Colt library (Colt Home Page) to produce random numbers. Colt has been widely used for scientific computation and is well validated. Repast makes its Colt-based random number sources accessible through a “random helper” class with a supporting internal registry. The internal registry supports efficient reuse of random streams. It also allows all random streams to be simultaneously reset when a user changes the default random seed.

ReLogo

ReLogo wraps much of the Repast Simphony library's functionality into a semantically simple but powerful package. The goal is to provide a fast track so modelers can quickly develop simulations that can later be scaled up, as needed, using the features of the full Repast Simphony library. A tool for automatically converting NetLogo models into ReLogo models is also included.

ReLogo semantics are drawn from the venerable Logo lineage of software (Harvey 1997), informed especially by the previously discussed StarLogo (Resnick 1996) and NetLogo (Wilensky 2012) variants, while maintaining the object-oriented nature of Repast Simphony.

ReLogo provides a streamlined Eclipse perspective to create Repast Simphony models with turtles, links, patches, and observers. An example is shown in the Innovation model discussed later. ReLogo models are programmed using a Logo-style syntax (the ReLogo domain-specific language) based on the Groovy (Groovy: A dynamic language for the Java platform) dynamic language. Groovy itself is a widely used, free, and open source Java Virtual Machine language that both compiles to Java bytecode and tightly integrates with Java. Thanks to its Groovy foundations, ReLogo freely interoperates with Groovy and Java. ReLogo programmers can use any Java or Groovy library without special syntax. They can also write Groovy or Java code fluidly at any point in a ReLogo file to gain access to the advanced features of either language.

A ReLogo model is compiled by using the Groovy Eclipse joint compiler. The ReLogo code is first converted into its Abstract Syntax Tree (AST) representation. The AST is then repeatedly transformed by Repast Simphony to generate model-specific custom ReLogo code, as needed. The end result is Java byte code, which is run as a regular Repast Simphony model.

Data collection

The Repast Simphony data collection system is designed to gather and store information from a simulation while it is running. The system typically collects predefined sets of data from each agent at each time step: a particular piece of logged data represents the state of the simulation at a particular time. This logged data can contain aggregate and nonaggregate values. An aggregate data value represents the result of an aggregate operation (e.g., summation) performed over some collection of individual values. Typically, these individual values correspond to agent properties. A nonaggregate value is some individual value, again typically an agent property.

Data collection in Repast Simphony consists of three phases, namely initialization, recording, and termination. Initialization and termination occur once per run, whereas recording occurs at some regular interval (e.g., once every three ticks). Crucial to the proper functioning of data collection is specification of a set of notifications associated with these phases. For example, when writing a row of data in tabular delimited format, it is necessary to notify the writer that all of the data for that row has been collected and can now be written. Similarly, a histogram component requires notification that the record phase has ended and that the histogram should now be produced with the recorded data.

The first Repast Simphony data collection system was built upon Log4J (LOG4J – Logging Services). Although Log4J is a widely used for fast, minimal overhead logging,

adapting it to the requirements of Repast Simphony's data collection functionality proved to be problematic. Log4J was difficult to adapt to the time-stamped and time-phased nature of Repast Simphony data collection. Log4J has no real user-facing mechanism for repeated initialization and termination from within a single application instance. Our incremental understanding of these requirements and adapting them to Log4J's system of logging levels and logging messages resulted in code that was inflexible and hard to manage. The isolation available in the multilayered Repast Simphony logging system made it easier for the Repast development team to transition from Log4J to custom code while largely maintaining backward compatibility.

Consequently, the data collection mechanism was replaced by a new one built on the following abstractions: data sources, data sinks, and data sets. Data sources are just that, sources of data. A nonaggregate data source receives a single object (e.g., an agent) and returns some data value, typically but not necessarily a property of that agent. An aggregate data source receives multiple objects, and performs some aggregate operation over all of the objects. It then returns the result of that operation. For example, it might calculate the mean value of some property over all of the objects. A data sink writes the output of these data sources, for example, to a file or a chart. A data set manages a collection of sources and sinks, providing the objects on which the data sources operate and directing the results to data sinks. Each data source can be thought of as a column in a tabular spreadsheet type of format, and each row is the result of recording data from each data source associated with a data set. Crucially, a data set also provides notifications to data sinks, notifying them that a new row of data is being started, that a row has ended, and that the entire record phase for the current time period has ended. It provides similar notifications to the data sources, for example, notifying an aggregate data source that the current row has started and thus any previous aggregate values should be reset. The new code embodying these abstractions is proving to be much more flexible than the previous Log4J code and has provided the basis for a more simple and more efficient user interface.

GIS

Geographic referencing is provided through geography projections that, like the other spatial projections in Repast Simphony, correlate the agents to positions in space. An agent's representation in a geography projection corresponds to a specific geographical feature, such as points, lines, and polygons. Repast Simphony uses GeoTools (GeoTools – The Open Source Java GIS Toolkit), an Open Geospatial Consortium (OGC) compliant library (OGC – Making location count), to provide support for the feature types described above, along with additional GIS data types and functions. Repast Simphony's GIS capabilities also include providing support for Environmental Systems Research Institute (ESRI) shapefiles (esri – Understanding our world) and a range of raster data files.

The geography projection is associated with a coordinate referencing system (CRS), which is based on the OGC standards and can be used to execute geographical queries on the topology of the agent features in the geography. Agents can query the geography to determine whether agent features overlap or are within a certain distance of, intersect with, or border other agent features in the geography. Features in the geography may have either static or dynamic positions. Typically, landmark features such as roads, buildings, and streams — represented by lines and polygons — are static. Mobile agent

feature types are typically represented by point features, whose latitude/longitude coordinates can be updated dynamically during the course of a simulation.

2D and 3D visualization

The Repast Simphony 2D and 3D visualization modules provide tools for interactively viewing running models in 2D and 3D. A 2D ReLogo example is shown in Figure 2. A 3D GIS example is shown in Figure 3.

The initial 2D visualization code was based on the Piccolo2D graphical framework (Piccolo2D – A Structured 2D Graphics Framework). Piccolo2D provides a 2D scene graph that allows for excellent animation, layout, zooming, and user interaction. Unfortunately, it proved to be too slow. In particular, it did not provide adequate performance for turtle and patch visualization in our new ReLogo language. For efficient visualization, the Repast Simphony design calls for a two-phased approach where (1) all the visual representations of all of the agents are updated to reflect the current states of the agents, and then (2) the entire scene is rendered. The intention is that the first phase performs no graphical screen updates. It is only after this first phase that the actual display is updated to reflect the changes made in the first phase.

The new 2D visualization mechanism in Repast Simphony 2.0 uses OpenGL (OpenGL – The Industry's Foundation for High Performance Graphics) via the Java Binding for the OpenGL (JOGL) API library (jogl – Java Binding for the OpenGL API) and implements the two-phased approach described above. The abstraction provided by the multilayered Repast Simphony visualization system simplified transitioning from Piccolo2D to OpenGL while largely maintaining backward compatibility. As before, agent objects do not draw themselves, but rather are associated with particular styles and layouts. These styles and layouts are then applied to each agent to determine its display location and its visual representation. The actual 2D visualization is accomplished by using a lightweight 2D scene graph implemented on top of OpenGL. The scene graph consists of a hierarchically arranged tree of nodes where operations on parent nodes apply to the children of those nodes. This new library also allows the user to easily create a variety of geometric shapes, create custom shapes, import Scalable Vector Graphics-based geometry, import images, and style these in a variety of ways. It supports object selection, zooming, and panning via a camera implementation. This new implementation has proven to be many times faster than the original Piccolo-based approach when rendering simulations of similar visual complexity.

Both the original Piccolo-based 2D and the new OpenGL-based 2D, as well as the current 3D implementation, were guided by the need for the same agent to be displayed simultaneously in a variety of topologies (networks, grids, and so on) and a variety of styles. For example, a single agent could be appear in a 2D display as a red circle in a grid layout and simultaneously in a 3D display as a blue cube participating in a network with the corresponding links visualized. Previous experience with agent visualization has shown that this capability is best accomplished by separating the agent's implementation from the visual representation of that agent.

In Repast Simphony, style and layout are implemented as Java interfaces. Both the 2D and 3D style interfaces consist of methods that receive an agent as a parameter and return some value, typically based on some property of the agent. For example, there

are methods that return the geometric representation (e.g., circle, cube, model-specific polygon) of the agent, the agent's color, rotation, scale, label, and so forth. An additional network style is used to style network edges. The layout interface works in a similar fashion. A location finding method takes an agent as a parameter and returns the location of that agent in either 2D or 3D space. Typical layouts compute the location for an individual agent relative to the topology in which it resides. The screen location of an agent in a network is determined by the incoming and outgoing links of that agent, whereas an agent in a grid topology is laid out according to its grid location. Consequently, the layout interface contains methods to associate it with the particular topology or topologies being displayed. Often, layouts need to be recalculated as agents are added and removed from the topology, and the layout interface reflects this need, as well. The implementation of style and layout as interfaces independent of agent implementation and the subsequent styling and layout of agents provide the required flexibility. Agents can be represented differently but simultaneously in 2D and 3D displays while participating in different types of topologies.

Both the 2D and 3D display implementations use a scene graph architecture. Individual agents are represented as geometric nodes in the scene. The appearance (color and so forth) of the object is determined by the style and applied to the object in some scene graph–specific way. For example, in a 3D display, the color of a node's surface appearance is determined by the color returned from the style. The node's ultimate screen location is the result of translating the node by the amount returned from the layout. Updates to the displays occur in two stages. Zooming, panning, and so forth are implemented as operations on the root node or camera of the scene. As mentioned above, display updates are a two-stage process. In the first, the scene graph is updated to reflect the latest styling and layout information, and then in the second, the scene graph is rendered.

GIS displays are available in both 2D and 3D implementations, both of which may use the same underlying geography projection. The 2D GIS display map is based on modified rendering utilities from GeoTools along with features from Piccolo2D. Agent features are visible in the 2D map according to the defined GIS style, which can be created and modified by the user at runtime via the built-in style editor, or imported from an OGC styled layer descriptor (SLD) file (OGC – Making Location Count: Standards). The 2D GIS display can be panned and zoomed in two dimensions.

An additional GIS display type is based on the NASA World Wind Java (WWJ) software development kit (SDK) (NASA World Wind) and shown in Figure 3. This WWJ display uses OpenGL to render an interactive globe on which geographical features are placed. The display behavior is similar to that of the 2D GIS display, such that features are represented by points, lines, and polygons and can be either static or dynamic. The default display contains several base raster layers for terrain and satellite data, which are fetched from public domain servers at runtime. Depending on the level of zoom magnification, finer and finer detail will be pulled automatically from the image servers. Agent feature layers are placed over the underlying raster image layers, and the extent of transparency, colors, and size of agent feature styles are fully customizable.

Freeze drying

Repast Simphony allows the user to serialize, or checkpoint, the state of the simulation for later retrieval and execution. Tabular and XML formats are supported. The tabular format

writes the serialized objects to rows and columns, where each row represents an object and each column a property of that object. An additional column contains a unique identifier generated by Repast Simphony for each object. Objects are serialized by type, with a single table holding the data for all objects of that type. Additional tables record the state of the object tree. For example, a Map may contain agents, and those agents may themselves contain additional objects. These tree tables would then record the parent–child relationships between the map and the agents, and the agents and the objects they contain.

The motivation for the tabular format was two-fold. First, we wanted to be able to load data into a variety of standard databases easily, and a tabular format made that relatively straightforward. Second, we wanted the state of agents to be editable offline. When saved to a delimited file format, the state of the simulation and its agents can be loaded into common tools, such as Microsoft Excel, and easily edited. Moreover, additional objects (e.g., agents) can be created offline by adding rows to the relevant table.

This tabular serialization format has worked reasonably well, although it does present some maintenance difficulties. In particular, large changes to Repast Simphony classes, such as networks, grids, and so forth that are serialized often require changes to the serialization code. The implementation of tabular serialization also exposed some differences in Java reflection implementations across operating systems.

XML format serialization was introduced as a simpler and more extensible serialization solution for those applications that did not need the benefits of the tabular format or had issues with it. XML serialization uses the XStream (XStream Home Page) serialization library to serialize objects to and from XML. The default serialization provided by XStream can be extended and customized by using its converter mechanism. Repast Simphony takes advantage of this capability by implementing custom converters for its major components, providing more efficient serialization. It also allows users to provide their own custom converters for serializing their own models. This functionality allows the user to handle or work around any issues in serialization without requiring any time-consuming changes to Repast Simphony itself.

Third-party applications

The Repast Simphony runtime interface's plug-in architecture is open to allow almost any kind of software to be added to the system. We have used this feature to provide users with a default set of built-in tools for model results analysis. Some of these tools are directly embedded within the runtime interface, and others are externally activated.

The default-embedded tools are generally Java-based libraries with Repast Simphony–compatible licenses. Examples include the JUNG network analysis system (JUNG – Java Universal Network/Graph Framework) and the SQL for Java Objects (JoSQL) library (JoSQL – What is JoSQL?). Repast uses JUNG for network storage and statistical analysis and uses JoSQL to allow users to query paused simulations interactively using SQL.

The default external tools are either not Java based or use licenses that are free and open source but incompatible with the Repast Simphony license. These external programs are activated by wizards that go through the following sequence: (1) ask users which data sets should be selected for analysis, (2) export the data to a format that is readable by the external tool, (3) note the licensing terms of the external program, (4) start the external executable, (5) automatically load the exported data into the target

application, and then (6) switch focus to the external program. The external tool loads into a separate memory space with no direct linkage to the initiating Repast model so there are no licensing conflicts. The experience is nonetheless seamless for the model user. Example default external tools include the R statistics system (http://www.r-project.org/), the GRASS GIS (GRASS GIS – The world's leading Free GIS software), the *ORA network analysis system (CASOS – *ORA), the Pajek network visualization system (Networks/Pajek: Program for Large Network Analysis), the iReport reporting tool (Jaspersoft Community – iReport Designer), the JasperReports reporting tools (Jaspersoft Community – JasperReports Library), common spreadsheets, the VisAD scientific visualization system (VisAD – McIDAS-V), and the Weka data mining system (WEKA – Machine Learning Group at the University of Waikato).

Results and discussion

CAS modeling example

This section uses a simple demonstration example to show how to develop models of complex adaptive systems with Repast Simphony. The approach presented here follows the methodology detailed in North and Macal (2007) and North and Macal (2009). The example scenario is intentionally simple so that it clearly shows the essential steps of the model design and development process. The scenario is an innovation diffusion environment in which a group of people randomly circulates. People occasionally communicate ideas when they encounter one another. When an idea is transferred, a new link is made between the transferring parties, and then any existing network links are forgotten. People occasionally develop new ideas. When they do so, they forget all of their old links. The quality of ideas is rated and tracked over time. The question posed to the simulation is: what is the quality of ideas over time?

In the sections that follow we discuss the appropriateness of using agent-based modeling to simulate the problem of interest, decompose the problem into components a using a series of design questions, and then show how the resulting implementation proceeds step-by-step. The combination of these steps produces a simple illustrative model.

When to use agent-based modeling

Agent-based modeling is a good choice compared to other modeling techniques when the problem naturally consists of agents or when the decision makers to be modeled meet the following criteria:

- Must have real individual behaviors;
- Adapt, change, or learn;
- Form dynamically changing relationships;
- Form organizations;
- Have spatial interactions;
- Have arbitrarily large populations; or
- When structural change is an output, not an input.

Our example satisfies two of the criteria: our innovation diffusion CAS is naturally composed of agents, and the agents form dynamically changing relationships.

Essential agent-based model design questions

Once agent-based modeling has been identified as a candidate methodology for solving the problem, the following questions can be asked:

- What are the modeling question(s)?
- Who are the stakeholders?
- What outputs are needed?
- What input data are available?
- What verification and validation (V&V) is needed?
- What alternative techniques where tried?
- Who are the agents?
- What are the agent behaviors?
- What is the agent environment?

Each of these questions will be briefly addressed in our next section for our example CAS.

What Are the Modeling Question(s)? The modeling question is: how does the quality of ideas change over time in the innovation network?

Who Are the Stakeholders? The stakeholders in this demonstration case are simply the modelers themselves.

What Outputs Are Needed? The output will be the history of the quality of innovation. This output will be displayed as a time series chart.

What Input Data Are Available? For this model, the input will be the probability of developing a new innovation on a given time step.

What V&V is Needed? The V&V step is key. It can be said that V&V turns toys into tools (North and Macal 2007). V&V is like a *properly prepared* court case in that modelers select a standard of evidence and then prepare arguments to meet the standard. To be properly prepared, countervailing facts must also be presented!

There are many sources of potential evidence for the arguments, including by:

- Conducting design and code walkthroughs;
- Performing unit testing;
- Conducting face validity checks;
- Docking against other models;
- Trying real-world and tricky artificial cases; and
- Conducting parameter sweeps and sensitivity analyses.

Simple face validity checks will be used for the innovation model.

What Alternative Techniques Were Tried? Many models of innovation diffusion have been developed (Meade and Islam 2006). Our model offers a simple, agent-based approach to this problem. Much more detail would be needed to translate this tutorial model into a full-scale model for studying innovation diffusion.

Who Are the Agents? The agents are individual people in the innovation network.

What Are the Agent Behaviors? The agents will randomly circulate on a continuous two-dimensional surface. When an agent with an innovation comes close enough to another agent, they will transfer their ideas. When an idea is transferred, a new link is made between the transferring parties. Any existing network links in the recipient will be deleted.

Agents develop at the rate given by the input innovation probability. When they do so, they delete all of their links.

Each idea is represented using a number. The value of the number represents the relative quality of the idea. The value also automatically maps into a display color for the agent and its links.

What Is the Agent Environment? The agent environment is a simple, continuous 2D surface with support for a network between agents.

Model implementation

Next we will illustrate how you can implement a Repast Simphony model. The model implementation follows directly from the design questions answered in the previous section.

To begin, you can create a new ReLogo project by clicking the new project wizard icon in the toolbar. This icon has a blue folder with red "RL" text in the upper left corner. The new project wizard dialog will appear with options for the project name and other advanced project options. The only required change in this dialog is the project name, which should be changed to "Innovation." You can click "Finish" to complete the new ReLogo project setup. Once the project setup process is complete, a new project named "Innovation" will appear in the left windowpane with a set of project components below. Two folders are visible — "src," which contains the model code; and "shape," which contains a set of optional icons that may be assigned to agents.

Expand the "src" folder to view the model components. The "src" folder contains the package "innovation.relogo," which itself contains the automatically generated model components such as the "UserObserver," "UserPatch," "UserTurtle," and others. The first step is to create a set of basic user controls that will appear in the model runtime window. You can double-click on the "UserGlobalsAndPanelFactory.groovy" file and modify the "addGlobalsAndPanelComponents" method as follows:

```
public void addGlobalsAndPanelComponents (){
  // Add the needed control buttons.
  addButton("setup")
  addToggleButton("go")
```

```
  // Add the model input.
  addInput("innovationProbability", 0.001)
}
```

The generated "UserGlobalsAndPanelFactory.groovy" file may contain additional example code that will not be activated given that it is commented. Commented code lines may begin with either "//" or "/*" and will be ignored by the simulation runtime. The three new lines added above create graphical buttons in the model runtime window that provide functions to setup and run the model. Changes to the "UserGlobalsAndPanelFactory.groovy" file are now complete. The "UserLink.groovy" file does not require any modifications at this time.

Next, you should open the "UserObserver.groovy" file, which will contain code that defines how to initialize the model and what happens at each simulation tick. Again, there will be example code in the generated "UserObserver" that can be ignored or deleted. The "UserObserver" requires two methods: "setup" and "go," which are listed below:

```
// Define the model builder.
def setup(){
  // Remove any existing turtles.
  clearAll()
  // Build a new set of turtles.
  createTurtles(20){
    setShape("person")
    setColor(gray())
    forward(1)
  }
  // Distribute an initial innovation.
  ask (turtles().first()) {
    innovation = yellow()
    setColor(innovation)
  }
}
// Define the model time step routine.
def go(){
  // Give each turtle a chance to act.
  ask(turtles()){ act() }
}
```

The "setup" method is called when the "setup" button in the runtime window is pressed. The button was previously created in the "UserGlobalsAndPanelFactory," and the functionality is provided in the "UserObserver." The setup routine first clears the model of any existing turtles using "clearAll()" for the case when the model had already been run. Next the "setup" method creates a set of 20 turtles, sets the shape of each turtle to the "person" icon shape, sets the initial color of the icon to gray, and moves the turtle agent one step forward. Finally, setup retrieves the first turtle and sets its

"innovation" parameter and color to yellow. The "go" method asks each turtle agent to execute its "act" method once each time the simulation is stepped. The turtle's "ask" method will be defined next. The "UserPatch.groovy" file does not require any additional modification.

You should open the "UserTurtle.groovy" file to edit the turtle agent class that will contain all of the Turtle agent behaviors. The "UserTurtle" agent behavior is defined entirely within the "act" method and "innovation" parameter, which are described below:

```
// Define the innovation tracker.
int innovation = 0
// Define the activity.
def act() {
  // Move forward randomly.
  left(random(15))
  right(random(15))
  forward(1.1)
  // Spontaneously create innovations.
  if (randomFloat(1.0) > (1.0-innovationProbability)) {
    // Develop the new innovation.
    innovation = 10 * random(11) + 25
    setColor(innovation)
    // Clear out my old links.
    ask (myLinks()) { die() }
  }
  // Check for an innovation.
  if (innovation > 0) {
    // Diffuse an innovation from "myself" to
    // nearby turtles ("self").
    ask (other(turtlesHere())) {
      // Get the innovation.
      innovation = myself().innovation
      setColor(myself().getColor())
      // Clear out the old links.
      ask (myLinks()) { die() }
      // Build a new link.
      UserLink newLink =
        createLinkWith(myself())
      newLink.setColor(myself().getColor())
    }
  }
}
```

The "UserTurtle" "act" method first moves the turtle forward by 1.1 units in a direction that is randomly determined by generating a random integer between 1 and 15 and then rotating the turtle left and right and finally moving 1.1 units forward. Next,

spontaneous innovations are randomly created within the turtle by generating a random integer between zero and one and checking this value against the input threshold value (1.0 – innovationProbability). If a new innovation is to be created, then the "innovation" value is set based on the random value, the turtle's icon color is set to the new innovation value, and the turtle's existing links are destroyed.

In the event that a new innovation is created, the turtle next asks for other turtle agents nearby (located in the same patch) to which the new innovation will be transferred. The innovating turtle sets its neighboring turtles' "innovation" and "color" to the new "innovation" value, destroys the neighboring turtles' existing links, and generates a new link between the turtle and its neighbor. Editing of all of the model code is now complete.

Next, you can add a "Total Innovation" chart. New charts begin with data sets. Data sets collect data from agents. As previously discussed, charts, files, or external analysis tools can use the resulting data. To begin creating the chart, you can right-click on the "Data Sets" item in the "Scenario Tree" in the lower-left panel of the Repast Simphony runtime interface. You should then type in "Total Innovation" as the "Data Set Id." You should then click "Next" and choose the "Method Data Sources" tab on the next wizard page. You can click "Add" to add a column to the data set. You should type in "totalInnovation" as the "Source Name," select "UserTurtle" as the "Agent Type," and "getInnovation" for the "Method." "Sum" should be automatically selected as the default aggregate function. The results of this step will direct Repast Simphony to add up all of the "innovation" values across all of the turtles for each time step. You should choose "Next" and then "Finish" to finalize the work. You can now proceed to add the chart itself.

To add the chart, you should right-click on the "Charts" item in the "Scenario Tree" in the lower left panel of the Repast Simphony runtime interface. You should then choose "Add Time Series Chart" from the popup menu. You should type in "Total Innovation" for the "Name" and then choose "Next." You should then click on the "totalInnovation" line on the chart properties wizard page and again choose "Next." You should type in "Total Innovation" for the chart title and "Innovation" for the Y-Axis title on the next wizard page and then click "Finish." To complete this step you, should save the chart definition by clicking on the floppy disk icon on the upper right corner of the Repast Simphony runtime interface.

To run the innovation model, the user can select the "run" icon from the toolbar. The run button has a green circular icon with a white triangle in the middle. The model runtime window will appear and can by initialized using the "Initialize Run" button in the toolbar. This button has a blue circular icon with a white circular "on" symbol in the middle. After the model has been initialized, the "User Panel" pane of the model runtime window will contain the "setup" and "go" buttons used to control the model. Clicking the "setup" button executes the "setup" method of the "UserObserver" described above, which creates an initial population of turtles in a circular arrangement in the center of the display panel on the right side of the simulation runtime window. One of the turtle icons is a different color than the others since the "UserObserver" set its "innovation" value to the nonzero value. Selecting the "go" button runs the model continuously. The turtles start to spread out in a circular pattern, and some additional turtles near the initial innovator may be colored according to the adopted "innovation." The runtime window for a model that was run continuously for several hundred time

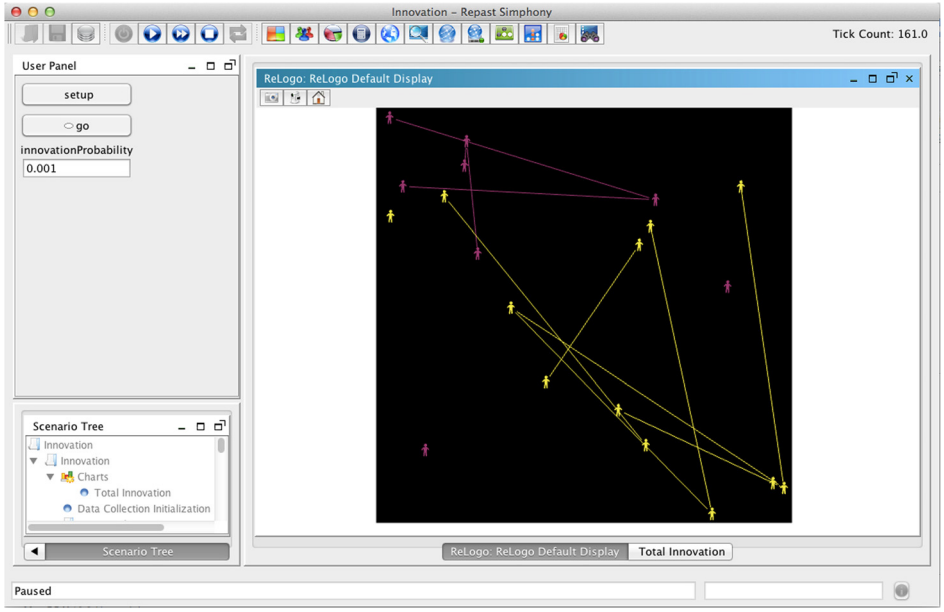

Figure 4 The example Repast Simphony innovation model.

steps is shown in Figure 4. Turtles that share the same "innovation" value are similar colors, and the links between these turtles indicate the "innovation" pedigree. Clicking on the "Total Innovation" chart tab on the lower right shows the resulting total innovation.

Conclusions

There are several directions for future work using the open source Repast Simphony agent-based environment for modeling in CASs, including to provide more scalable support for large-scale modeling; to simplify model specification; and to provide more facilities to support model verification and validation. In terms of scalability, the Repast team has recently completed a C++ Message Passing Interface implementation of the core Repast feature set that targets supercomputer execution. This new library, Repast for High Performance Computing (Collier and North 2011; Collier and North 2999), has recently been shown to have excellent weak scalability on Argonne National Laboratory's IBM Blue Gene/P.

Abbreviations

2D: Two-dimensional; 3D: Three-dimensional; API: Application programming interface; AST: Abstract syntax tree; BSD: Berkeley software distribution; CAS: Complex adaptive system; CRS: Coordinate referencing system; EndEC: Endogenous emergence of coordination; ESRI: Environmental systems research institute; GIS: Geographic information system; GPL: General public license; GRASS: Geographic resources analysis support system; JOGL: Java binding for the OpenGL; JoSQL: SQL for Java Objects; JPF: Java plug-in framework; JUNG: Java Universal Network/Graph Framework; MADCABS: Monitoring analysis, and diagnosis of distributed processes with agent-based systems; NQP: Not Quite Python; OGC: Open geospatial consortium; P&G: Procter & gamble; Repast: REcursive porous agent simulation toolkit; ROAD: Repast organization for architecture and design; SAF: Simphony application framework; SDK: Software development kit; SLD: styled layer descriptor; SQL: Structured query language; V&V: Verification and validation; WWJ: World wind Java; XML: Extensible markup language.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

MJN, NTC, JO, ERT, MB, and others developed the Repast Simphony code. MJN, NTC, JO, ERT, and MB wrote parts of the paper. MJN wrote the innovation demonstration model. All authors MJN, NTC, JO, ERT, CMM, MB, and PS read and approved the final manuscript.

Acknowledgments
The submitted manuscript has been create by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.
The authors also wish to acknowledge the many developers, supporters, and contributors to the Repast family of agent-based modeling software.

Received: 23 November 2012 Accepted: 17 January 2013
Published: 13 March 2013

References

Artel A, Teymour F, North MJ, Cinar A: **A multi-agent approach using perceptron-based learning for robust operation of distributed chemical reactor networks.** *Int Sci J Eng App Artif Intell* 2011, **24:**1035–1045.
Bonabeau E: **Agent-based modeling: Methods and techniques for simulating human systems**. In *Proceedings of the National Academy of Sciences*, Volume 99(3). Washington, D.C. USA: National Academy of Sciences Press; 2002:7280–7287.
Bozada T, Perkins T, North MJ, Simunich KL, Tatara E: **An applied approach to representing human behavior in military logistics operations**. In *Proceedings of the fall simulation interoperability workshop; September 10–15, 2006*. Edited by Weber R. Orlando, FL USA: Curran Associates; 2006:669–678.
Branting S, Wu Y, Srikrishnan R, Altaweel MR, SHULGI: **A geospatial tool for modeling human movement and interaction**. In *Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence*. Edited by North M, Macal C, Sallach D. Argonne, IL: Argonne National Laboratory; 2007:258–273.
CASOS – *ORA: 2999,: http://www.casos.cs.cmu.edu/projects/ora/.
Chiba S, Nishizawa M: **An easy-to-use toolkit for efficient java bytecode translators**. In *Proc. of 2nd international conference on generative programming and component engineering (GPCE '03). Springer Lecture Notes in Computer Science 2830*. Edited by Pfenning F, Smaragdakis Erfurt Y. Germany: Springer-Verlag; 2003:364–376.
Collier NT, North MJ: **Repast HPC: A library for large-scale agent-based modeling**. In *Large-Scale Computing Techniques for Complex System Simulations*. Edited by Dubitzky W, Kurowski K, Schott B. Hoboken, NJ: Wiley-IEEE Computer Society Press; 2011.
Collier NT, North MJ: **Parallel agent-based programming with Repast for High Performance Computing.** *Simulation* 2999, in press.
Collier N, Howe T, North MJ: **Onward and upward: The transition to Repast 2.0**. In *Proceedings of the first annual North American Association for Computational Social and Organizational Science conference*. Edited by Carley K. Pittsburgh: Carnegie Mellon University; 2003. Electronic Proceedings.
Colt Home Page: 2999, http://acs.lbl.gov/software/colt/.
Eclipse Home Page: 2999, http://www.eclipse.org/.
ESRI – Understanding our world: 2999, http://www.esri.com.
Feurzeig W, Papert S, Bloom M, Grant R, Solomon C: **Programming-languages as a conceptual framework for teaching mathematics.** *Interface* 1970, **4:**#2.
GeoTools – The Open Source Java GIS Toolkit: 2999, http://geotools.org/.
GRASS GIS – The world's leading Free GIS software: 2999, http://grass.fbk.eu/.
Groovy: A dynamic language for the Java platform: 2999, http://groovy.codehaus.org/.
Harvey B: *Computer Science Logo Style*. Boston: MIT Press; 1997.
Holland JH: **Studying complex adaptive systems.** *J Syst Sci Complex* 2006, **19:**1–8.
Jaspersoft Community – iReport Designer: 2999, http://jasperforge.org/projects/ireport.
Jaspersoft Community – JasperReports Library: 2999, http://jasperforge.org/projects/jasperreports.
Java Plug-in Framework (JPF) Project: 2999, http://jpf.sourceforge.net/.
jogl – Java Binding for the OpenGL API: 2999, http://jogamp.org/jogl/www/.
JoSQL – What is JoSQL?: 2999, http://josql.sourceforge.net/.
JUNG – Java Universal Network/Graph Framework: 2999, http://jung.sourceforge.net/.
Ke J, Holland JH: **Language origin from an emergentist perspective.** *Applied Linguistics* 2006, **27:**691–716.
LOG4J – Logging Services: 2999. http://logging.apache.org/log4j/.
Luke S, Cioffi-Revilla C, Panait L, Sullivan K, Balan G: **MASON: A multiagent simulation environment.** *SIMULATION* 2005, **81:**517–527.
Macal CM: **Agent-based Modeling and Artificial Life**. In *Encyclopedia of Complexity and System Science*. Edited by Meyers RA.: Springer; 2009:112–131. ISBN 978-0-387-75888-6.
Macal CM, North MJ: **Tutorial on agent-based modeling and simulation.** *J Simul* 2010, **4:**151–162.
Mahalik MR, Conzelmann G, Stephan CH, Mintz MM, Veselka TD, Tolley GS, Jones DW: **Modeling the transition to hydrogen-based transportation**. In *Proceedings of the agent 2007 conference on complex interaction and social emergence*. Edited by North M, Macal C, Sallach D. Argonne, IL: Argonne National Laboratory; 2007.
Meade N, Islam T: **Modelling and forecasting the diffusion of innovation – A 25-year review.** *Int J Forecast* 2006, **22:**519–545.
Minar N, Burkhart R, Langton C, Askenazi M: *The Swarm simulation system: A toolkit for building multi-agent simulations*. Santa Fe: Santa Fe Institute; 1996. Working Paper 96-06-042.
NASA World Wind: 2999, http://worldwind.arc.nasa.gov/java/.
NetLogo User Manual – FAQ (Frequently Asked Questions): 2999, http://ccl.northwestern.edu/netlogo/faq.html.
Networks/Pajek: Program for Large Network Analysis: 2999, http://vlado.fmf.uni-lj.si/pub/networks/pajek/.
North MJ, Macal CM: *Managing business complexity: Discovering strategic solutions with agent-based modeling and simulation*. Oxford, U.K.: Oxford University Press; 2007.

North MJ, Macal CM: **Foundations of and recent advances in artificial life modeling with Repast 3 and Repast Simphony**. In *Artificial Life Models in Software*. 2nd edition. Edited by Adamatzky A, Komosinski M. Heidelberg: Springer; 2009:37–60.
North MJ, Howe TR, Collier NT, Vos RJ: **The Repast Simphony runtime system**. In *Proceedings of the agent 2005 conference on generative social processes, models, and mechanisms*. Edited by Macal C, North M, Sallach D. Argonne, IL: Argonne National Laboratory; 2005:151–158.
North MJ, Collier NT, Vos RJ: *Experiences creating three implementations of the Repast agent modeling toolkit*, ACM Transactions on Modeling and Computer Simulation. 16(1)th edition. New York: ACM; 2006:1–25.
North MJ, Howe TR, Collier NT, Vos JR: **A declarative model assembly infrastructure for verification and validation**. In *Advancing social simulation: the first world congress*. Edited by Takahashi S, Sallach DL, Rouchier J. Heidelberg: Springer; 2007:129–140.
North MJ, Macal CM, St. Aubin J, Thimmapuram P, Bragen M, Hahn J, Karr J, Brigham N, Lacy ME, Hampton D: **Multi-scale agent-based consumer market modeling.** *Complexity* 2010, **15**(5):37–47.
OGC – Making Location Count: Standards: 2999, http://www.opengeospatial.org/standards/sld.
OGC – Making location count: 2999, http://www.opengeospatial.org/.
OpenGL – The Industry's Foundation for High Performance Graphics: 2999, http://www.opengl.org/.
Ozik J, North MJ: **Modeling endogenous coordination with a dynamic language**. In, Simulating interacting agents and social phenomenon: the second world congress on social simulationEdited by Takadama K, Cioffi-Revilla C, Deffuant G. Heidelberg, FRG: Springer; 2010:265–276.
Parker MT: **Ascape: Abstracting complexity.** *Nat Res Environ Issues* 2001, **8:**21–30.
Piccolo2D – A Structured 2D Graphics Framework: 2999, http://www.piccolo2d.org/.
Repast – The Repast Suite: 2999. http://repast.sourceforge.net/.
Resnick M: **StarLogo: An environment for decentralized modeling and decentralized thinking**. In *Conference companion on human factors in computing systems: common ground*. Edited by Tauber MJ. New York: ACM; 1996:11–12.
Sallach D, Macal C: **The simulation of social agents: An introduction.** *Soc Sci Comput Rev* 2001, **19**(3):245–248.
Standish RK: **Going stupid with EcoLab.** *Simulation* 2008, **84:**611–618.
Swarm: license: 2999, http://www.swarm.org/index.php/Swarm:_license.
Tao G, Minett JW, Jinyun K, Holland JH, Wang WSY: **Coevolution of lexicon and syntax from a simulation perspective: Research articles.** *Complexity* 2005, **10:**50–62.
Tatara E, North MJ, Hood CS, Teymour F, Cinar A: **Agent-based control of spatially distributed chemical reactor networks**. In *Engineering self-organising systems: third international workshop revised selected papers*, Lecture Notes in Computer Science Series, Volume 3910. Edited by Brueckner SA, DiMarzo Serugendo G, Hales D, Zambonelli F. Heidelberg: Springer; 2006:222–231.
Tisue S, Wilensky U: **NetLogo: Design and implementation of a multi-agent modeling environment**. In *SwarmFest*. Edited by Riolo R. Ann Arbor, MI: Swarm Development Group; 2004.
VisAD – McIDAS-V: 2999, http://www.ssec.wisc.edu/~billh/visad.html.
WEKA – Machine Learning Group at the University of Waikato: 2999, http://www.cs.waikato.ac.nz/ml/weka/.
Wilensky U: *NetLogo. Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University*. 2012.
XStream Home Page: 2999, http://xstream.codehaus.org/.

doi:10.1186/2194-3206-1-3
Cite this article as: North *et al.*: Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling* 2013, **1**:3.

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Immediate publication on acceptance
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com